1. 上课约定须知

课程主题：Flink 源码解析 -- 第一次课（集群主节点 JobManager 启动）
上课时间：20:00 - 23:00
课件休息：21:30 左右 休息10分钟
课前签到：如果能听见音乐，能看到画面，请在直播间扣 666 签到

2. 本次内容大纲

FLink 源码分析大概会有 5-6 次课，主要内容大致如下：

```
01、Flink RPC 分析
02、Flink 集群启动的 shell 脚本分析
03、集群主节点 JobManager 启动分析
04、集群主节点 TaskManager 启动分析
05、Flink Job 的构建和提交（四层图架构）
06、Flink Job 具体提交（client 如何提交的）
07、Flink Job 具体提交（server 端是如何处理）
08、申请 slot 是怎么做的 + Slot的管理（申请 + 释放）
09、StreamTask 的部署
10、MailBox 线程模型
11、StreamTask 执行
12、Flink 状态管理
13、Flink Checkpoint 机制
```

今天是 Flink 源码的第一次课程，主要讲解的是 Flink 的集群主节点启动，主要内容分为以下六点：

```
1、Flink RPC 剖析
2、Flink 集群启动脚本分析
3、Flink 集群启动 JobManager 启动源码剖析
    4、WebMonitorEndpoint 启动和初始化源码剖析
    5、ResourceManager 启动和初始化源码剖析
    6、Dispatcher 启动和初始化源码剖
```

3. 详细课堂内容

3.1. Flink RPC 详解

大数据技术栈中的技术组件非常丰富，大致总结一下各大常见组件的 RPC 实现技术：

技术组件	RPC 实现
Hadoop	NIO + Protobuf
HBase	HBase-2.x 以前：NIO + ProtoBuf HBase-2.x 以后：Netty
ZooKeeper	BIO + NIO + Netty
Spark	Spark-1.x 基于 Akka Spark-2.x 基于 Netty
Flink	Akka + Netty

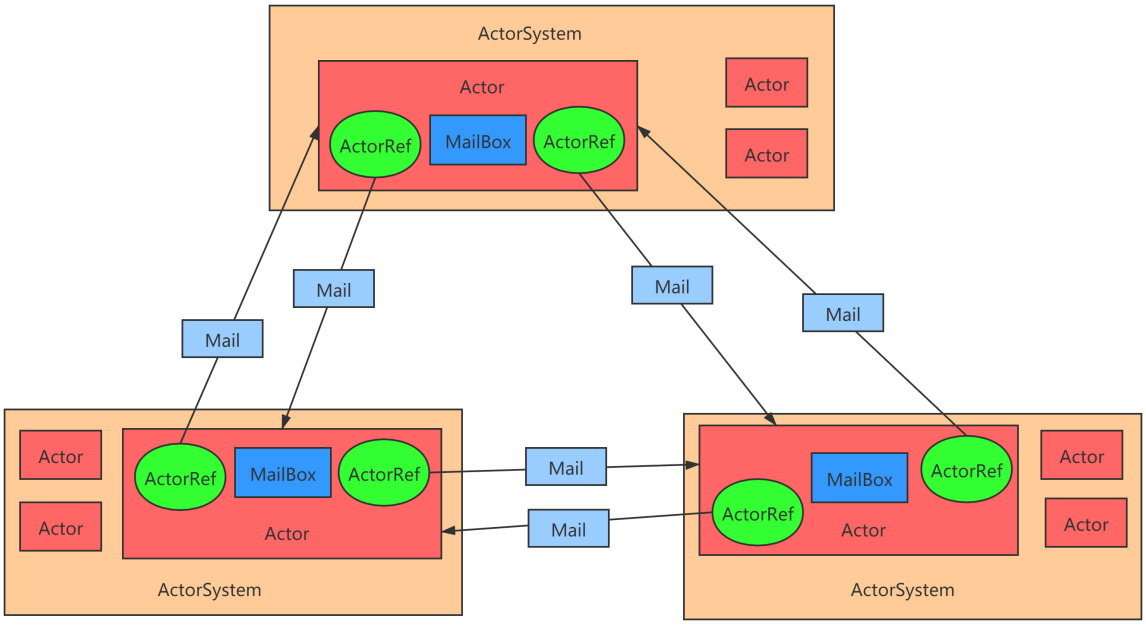
总结：Flink 的 RPC 实现：基于 Scala 的网络编程库： Akka

Akka 的特点总结：

```
1、它是对并发模型进行了更高的抽象；
2、它是异步、非阻塞、高性能的事件驱动编程模型；
3、它是轻量级事件处理（1GB 内存可容纳百万级别个 Actor）；
```

几个储备知识：关于对 Akka 的 ActorSystem 和 Actor 的理解：

```
1、ActorSystem 是管理 Actor 生命周期的组件，Actor 是负责进行通信的组件
2、每个 Actor 都有一个 MailBox，别的 Actor 发送给它的消息都首先储存在 MailBox 中，通过这种方式可以实现异步通信。
3、每个 Actor 是单线程的处理方式，不断的从 MailBox 拉取消息执行处理，所以对于 Actor 的消息处理，不适合调用会阻塞的处理方法。
4、Actor 可以改变他自身的状态，可以接收消息，也可以发送消息，还可以生成新的 Actor
5、每一个ActorSystem 和 Actor都在启动的时候会给定一个 name，如果要从ActorSystem中，获取一个 Actor，则通过以下的方式来进行 Actor的获取：
akka.tcp://actorsystem_name@bigdata02:9527/user/actor_name 来进行定位
6、如果一个 Actor 要和另外一个 Actor进行通信，则必须先获取对方 Actor 的 ActorRef 对象，然后通过该对象发送消息即可。
7、通过 tell 发送异步消息，不接收响应，通过 ask 发送异步消息，得到 Future 返回，通过异步回到返回处理结果。
```



关于 Actor 类似的概念：

```
1、HDFS ： proxy
2、Akka ： ActorRef
3、Flink ： XXXGateWay
```

在阅读 Flink 源码过程中，如果你见到有这种类型的代码，其实就是在发送 RPC 请求

```
// resourceManagerGateway 就可以理解成： 当前节点中，对于 ResourceManager 代理对象的封装
resourceManagerGateway.requestSlot();
// 代码跳转到：resourceManager.requestSlot();
```

总结： Flink 中的 RpcEndpoint ，在作用上，等同于 Akka 中的 Actor

Spark 的 RPC 实现虽然是为了替换 Akka 而诞生的，但是它实际上可以看成一个简化版的 Akka，仍然遵循许多 Actor Model 的抽象。例如

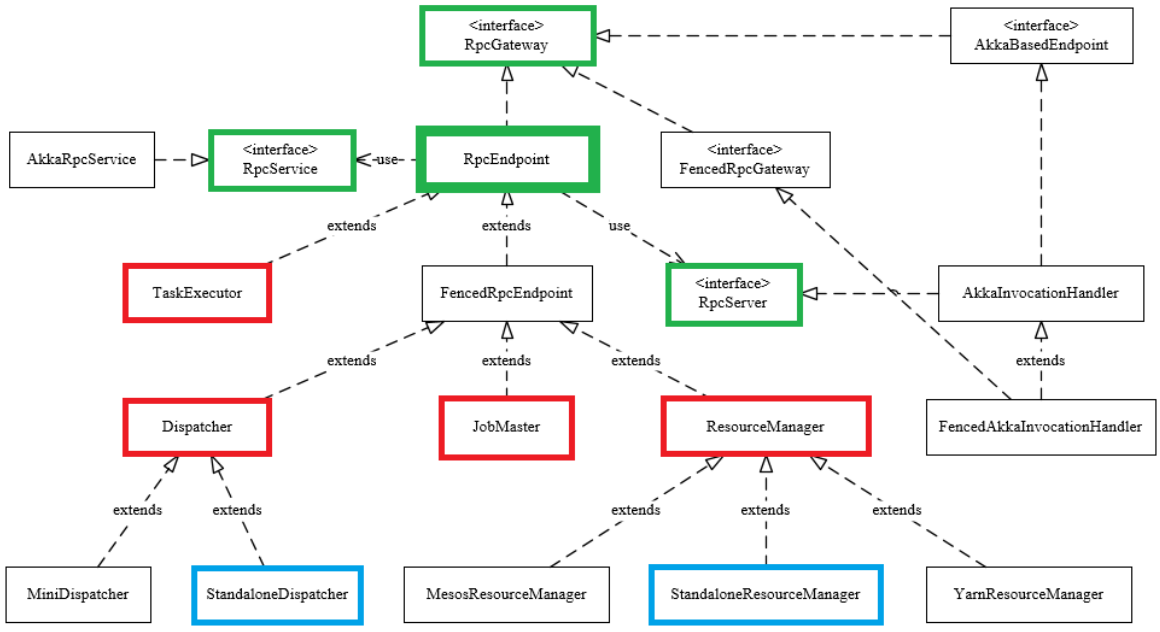
- RpcEndpoint 对应 Actor
- RpcEndpointRef 对应 ActorRef
- RpcEnv 对应 ActorSystem

Flink 中的 RPC 实现主要在 `flink-runtime` 模块下的 `org.apache.flink.runtime.rpc` 包中，涉及到的最重要的 API 主要是以下这四个：

组件	意义
RpcGateway	用于远程调用的代理接口。 RpcGateway 提供了获取其所代理的 RpcEndpoint 的地址的方法。在实现一个提供 RPC 调用的组件时，通常需要先定一个接口，该接口继承 RpcGateway 并约定好提供的远程调用的方法。
RpcServer	相当于 RpcEndpoint 自身的的代理对象（self gateway)。RpcServer 是 RpcService 在启动了 RpcEndpoint 之后返回的对象，每一个 RpcEndpoint 对象内部都有一个 RpcServer 的成员变量，通过 getSelfGateway 方法就可以获得自身的代理，然后调用该 Endpoint 提供的服务。
RpcEndpoint	对 RPC 框架中提供具体服务的实体的抽象，所有提供远程调用方法的组件都需要继承该抽象类。另外，对于同一个 RpcEndpoint 的所有 RPC 调用都会在同一个线程（RpcEndpoint 的“主线程”）中执行，因此无需担心并发执行的线程安全问题。
RpcService	是 RpcEndpoint 的运行时环境，RpcService 提供了启动 RpcEndpoint , 连接到远端 RpcEndpoint 并返回远端 RpcEndpoint 的代理对象等方法。此外，RpcService 还提供了某些异步任务或者周期性调度任务的方法。

简单总结：

```
1、RpcGateway     路由，RPC的老祖宗，各种其他RPC组件，都是 RpcGateWay 的子类
2、RpcServer      RpcService（ActorSystem）和 RpcEndpoint（Actor）之间的粘合层
3、RpcEndpoint    业务逻辑载体，对应的 Actor 的封装
4、RpcService     对应 ActorSystem 的封装
```



最终总结一下：RpcEndpoint 下面有四个比较重要的子类：

```
1、TaskExecutor 集群中从节点中最重要的角色，负责资源管理
2、Dispatcher 主节点中的一个工作角色，负责 job 调度执行
3、JobMaster 应用程序中的主控程序，类似于 Spark 中的 Driver 的作用，或者 MapReduce 中的 ApplicationMaster
4、ResourceManager 集群中的主节点 JobManager 中的负责资源管理的角色，和 TaskExecutor 一起构成资源管理的主从架构
```

当在任意地方发现要创建这四个组件的任何一个组件的实例对象的时候，创建成功了之后，都会要去执行他的 onStart() ，因为他们都是 RpcEndpoint 的子类，在集群启动的源码分析中，其实这些组件的很多的工作流程，都被放在 onStart() 里面。

3.2. 阅读源码的准备

到底需要那些准备?

```
1、技术组件的大致工作原理
2、选择版本
3、搭建源码阅读的环境
4、储备知识 + 场景驱动
```

3.3. Flink 集群启动脚本分析

Flink 集群的启动脚本在：flink-dist 子项目中，位于 flink-bin 下的 bin 目录：启动脚本为：start-cluster.sh

该脚本会首先调用 config.sh 来获取 masters 和 workers，masters 的信息，是从 conf/masters 配置文件中获取的， workers 是从 conf/workers 配置文件中获取的。然后分别：

```
1、通过 jobmanager.sh 来启动 JobManager
2、通过 taskmanager.sh 来启动 TaskManager
```

它们的内部，都通过 flink-daemon.sh 脚本来启动 JVM 进程，分析 flink-daemon.sh 脚本发现：

```
1、JobManager 的启动代号：standalonesession，实现类是：
StandaloneSessionClusterEntrypoint
2、TaskManager 的启动代号：taskexecutor，实现类是：TaskManagerRunner
```

最终通过 java 命令来启动对应的 JVM 进程!

HDFS 集群启动的 shell 编写的方式也是一样的：

```
1、start-all.sh / start-dfs.sh
2、hadoop-daemon.sh start namenode/datanode/zkfc/journalnode
3、java org.apache.hadoop.server.namenode.NameNode
```

3.4. Flink 主节点 JobManager 启动分析

Flink 主从架构： 主节点： JobManager + 从节点： TaskManager

JobManager 是 Flink 集群的主节点，它包含三大重要的组件：

```
1、ResourceManager
    Flink的集群资源管理器，只有一个，关于slot的管理和申请等工作，都由他负责

2、Dispatcher
    负责接收用户提交的 JobGragh，然后启动一个 JobMaster， 类似于 YARN 集群中的 AppMaster 角色，类似于 Spark Job 中的 Driver 角色
    内部有一个持久服务：JobGraghStore

3、WebMonitorEndpoint rest服务 flink run
    里面维护了很多很多的Handler，如果客户端通过 flink run 的方式来提交一个 job 到 flink 集群，最终，
    是由 WebMonitorEndpoint 来接收，并且决定使用哪一个 Handler 来执行处理
    例如：submitJob ===> JobSubmitHandler
```

当你提交一个 Job 到 Flink 集群运行的时候：

```
4、JobMaster/JobManager
    负责一个具体的 Job 的执行，在一个集群中，可能会有多个 JobManager 同时执行
    类似于 YARN 集群中的 AppMaster 角色，类似于 Spark Job 中的 Driver 角色
    由 createJobManagerRunner() 创建实现
```

关于 JobManager 的区分：

```
1、如果我们将 FLink 是主从架构，那么这个 JobManager 就是指主节点，它包含上面讲述的三种角色：ResourceManager，Dispatcher，WebMonitorEndpoint

2、如果我们将 Job 提交到 YARN 运行的时候，事实上，可以通过启动一个小集群的方式来运行，这个小集群的主节点也是 JobManager，这就是 Flink on YARN 的 Session 模式。你把 job 提交到 YARN 运行的时候，有三种模式：per-job, session, application
```

总结一下：

```
Flink 集群的主节点内部运行着：ResourceManager 和 Dispatcher，当 client 提交一个 job 到集群运行的时候（客户端会把该 Job 构建成一个 JobGragh 对象），主节点接收到提交 job 的 rest 请求之后，WebMonitorEndpoint 执行处理：会通过 Router 进行解析找到对应的 Hanlder 来执行处理，处理完毕之后，转交给 Dispatcher 来处理，Dispatcher 负责拉起 JobMaster 来负责这个 Job 内部的 Task 的部署执行，执行 Task 所需要的资源，JobMaster 向 ResourceManager 申请。
```

根据以上的启动脚本分析：JobManager 的启动主类：StandaloneSessionClusterEntrypoint

```
// 入口，解析命令行参数 和 配置文件 flink-conf.yaml
StandaloneSessionClusterEntrypoint.main()

    ClusterEntrypoint.runClusterEntrypoint(entrypoint);

        // 启动插件组件，配置文件系统实例等
        clusterEntrypoint.startCluster();

            runCluster(configuration, pluginManager);

                // 第一步：初始化各种服务（8个基础服务）
                initializeServices(configuration, pluginManager);
```

```
                // 创建 DispatcherResourceManagerComponentFactory，初始化各种组件的
工厂实例
                // 其实内部包含了三个重要的成员变量：
                // 创建 ResourceManager 的工厂实例
                // 创建 Dispatcher 的工厂实例
                // 创建 WebMonitorEndpoint 的工厂实例
                createDispatcherResourceManagerComponentFactory(configuration);

                // 创建 集群运行需要的一些组件：WebMonitorEndpoint，Dispatcher，
ResourceManager 等
                // 创建和启动 ResourceManager
                // 创建和启动 Dispatcher
                // 创建和启动 WebMonitorEndpoint
                clusterComponent =
dispatcherResourceManagerComponentFactory.create(...)
```

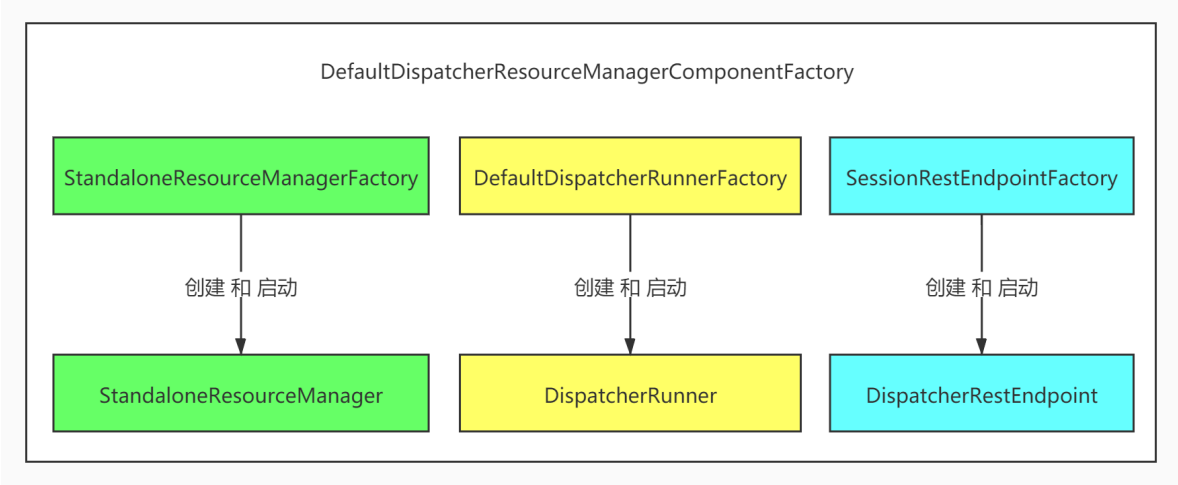
第一步 initializeServices() 中做了很多服务组件的初始化：

```
// 初始化和启动 AkkaRpcService，内部其实包装了一个 ActorSystem
commonRpcService = AkkaRpcServiceUtils.createRemoteRpcService(...)

// 启动一个 JMXService，用于客户端链接 JobManager JVM 进行监控
JMXService.startInstance(configuration.getString(JMXServerOptions.JMX_SERVER_POR
T));

// 初始化一个负责 IO 的线程池
ioExecutor = Executors.newFixedThreadPool(...)

// 初始化 HA 服务组件，负责 HA 服务的是：ZooKeeperHaServices
haServices = createHaServices(configuration, ioExecutor);

// 初始化 BlobServer 服务端
blobServer = new BlobServer(configuration, haServices.createBlobStore());
blobServer.start();

// 初始化心跳服务组件，heartbeatServices = HeartbeatServices
heartbeatServices = createHeartbeatServices(configuration);

// 启动 metrics（性能监控） 相关的服务，内部也是启动一个 ActorSystem
MetricUtils.startRemoteMetricsRpcService(configuration,
commonRpcService.getAddress());

// 初始化一个用来存储 ExecutionGraph 的 Store，实现是：
FileArchivedExecutionGraphStore
archivedExecutionGraphStore = createSerializableExecutionGraphStore(...)
```

第二步 createDispatcherResourceManagerComponentFactory(configuration) 中负责初始化了很多组件的工厂实例：

```
1、DispatcherRunnerFactory，默认实现：DefaultDispatcherRunnerFactory，生产
DefaultDispatcherRunner
2、ResourceManagerFactory，默认实现：StandaloneResourceManagerFactory，生产
StandaloneResourceManager
3、RestEndpointFactory，默认实现：SessionRestEndpointFactory，生产
DispatcherRestEndpoint
```

关于 DefaultDispatcherResourceManagerComponentFactory 这个组件工厂，它的内部组成：

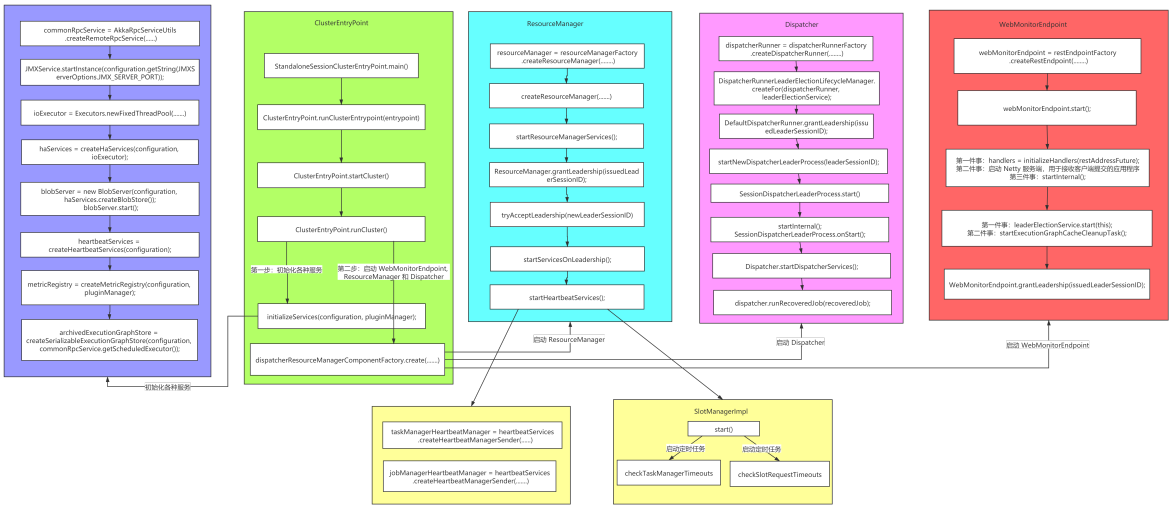


其中，DispatcherRunnerFactory 内部也实例化了一个：
SessionDispatcherLeaderProcessFactoryFactory 组件

第三步 dispatcherResourceManagerComponentFactory.create(...) 中主要去创建 三个重要的组件：

```
1、DispatcherRunner，实现是：DefaultDispatcherRunner
2、ResourceManager，实现是：StandaloneResourceManager
3、WebMonitorEndpoint，实现是：DispatcherRestEndpoint
```

最终总结一下：



3.5. WebMonitorEndpoint 启动和初始化源码剖析

核心入口：

```
DispatcherResourceManagerComponentFactory.create(....)
```

第一件事：

```
// 初始化 WebMonitorEndpoint
webMonitorEndpoint = restEndpointFactory.createRestEndpoint(
                        configuration, dispatcherGatewayRetriever,
resourceManagerGatewayRetriever,
                        blobServer, executor, metricFetcher,

 highAvailabilityServices.getClusterRestEndpointLeaderElectionService(),
                        fatalErrorHandler);

// 启动 WebMonitorEndpoint
webMonitorEndpoint.start();
```

核心流程：

1. 初始化一大堆 Handler
2. 启动一个 Netty 的服务端，注册这些 Handler
3. 启动内部服务：执行竞选！WebMonitorEndpoint 本身就是一个 LeaderContender 角色。
4. 竞选成功，其实就只是吧 WebMontiroEndpoint 的 address 以及跟 zookeeper 的 sessionID 写入到 znode 中

3.6. ResourceManager 启动和初始化源码剖析

核心入口：

```
DispatcherResourceManagerComponentFactory.create(....)
```

第二件事：

```
// 初始化 ResourceManager
resourceManager = resourceManagerFactory.createResourceManager(
                        configuration, ResourceID.generate(),
                        rpcService, highAvailabilityServices, heartbeatServices,
                        fatalErrorHandler, new ClusterInformation(hostname,
blobServer.getPort()),
                        webMonitorEndpoint.getRestBaseUrl(), metricRegistry,
hostname
                    );

// 启动 ResourceManager
resourceManager.start();
```

总结：

```
1、它是 RpcEndpoint 的子类， 关注 onStart()
2、它是 LeaderContender 的子类，所以要关注：选举
3、启动 ResourceManagerService: 两个心跳服务，两个定时服务
```

3.7. Dispatcher 启动和初始化源码剖

核心入口：

```
DispatcherResourceManagerComponentFactory.create(....)
```

第三件事：

```
// 初始化 并启动 DispatcherRunner
dispatcherRunner = dispatcherRunnerFactory.createDispatcherRunner(

 highAvailabilityServices.getDispatcherLeaderElectionService(),
fatalErrorHandler,
                        // TODO_MA 注释： 注意第三个参数
                        new
HaServicesJobGraphStoreFactory(highAvailabilityServices),
                        ioExecutor, rpcService, partialDispatcherServices
                    );

dispatchper = createDispatcher();
dispatchper.start();
```

总结：

```
1、启动 JobGraphStore 服务
2、从 JobGraphStrore 恢复执行 Job，要启动 Dispatcher
```

4. 本次课程总结

本次课程，主要讲解集群的启动，今天主要讲解的是主节点 JobManager 的启动，在启动过程中，会有各种服务组件的初始化工作

```
1、Flink RPC 剖析
2、Flink 集群启动脚本分析
3、Flink 集群启动 JobManager 启动源码剖析
    4、WebMonitorEndpoint 启动和初始化源码剖析
    5、ResourceManager 启动和初始化源码剖析
    6、Dispatcher 启动和初始化源码剖
```

一定要注意，Flink Standalone 集群的主节点 JobManager 的内部包含非常重要的三大组件，在启动过过程中，会依次启动，这三大组件分别是：

```
1、WebMonitorEndpoint
2、ResourceManager
3、Dispatcher
```

5. 本次课程作业

使用 Flink RPC 组件模拟实现 YARN，这个需求和我之前在讲 Spark 源码的时候，讲解的使用 Akka 模拟实现 YARN 的需求是类似的，只不过需要使用的技术是 Flink RPC 组件！

实现要求：

1、资源集群主节点叫做： ResourceManager，负责管理整个集群的资源
2、资源集群从节点叫做： TaskExecutor，负责提供资源
3、ResourceManager 启动的时候，要启动一个验活服务，制定一种机制（比如：某个 TaskExecutor 的连续5次心跳未接收到，则认为该节点死亡）实现下线处理
4、TaskExecutor 启动之后，需要向 ResourceManager 注册，待注册成功之后，执行资源（按照 Slot 进行抽象）汇报 和 维持跟主节点 ResourceManager 之间的心跳以便 ResourceManager 识别到 TaskExecutor 的存活状态